

# Graphics for Free

**Martin C. Carlisle**

Computer Science Department  
US Air Force Academy, CO 80840-6234  
mcc@cs.usafa.af.mil

## 1 Introduction

Students find computer graphics one of the most interesting topics in computer science. Unfortunately, writing programs with graphics requires understanding concepts that are usually beyond the scope of an introductory computer science course. For example, in Windows 95, a program that uses graphics must have an event loop that dispatches messages to the appropriate handler. Event loops, messages and handlers are well beyond the grasp of someone just learning about variables! As a result, programming assignments for introductory courses tend to use no graphics, or simple ANSI graphics (see, e.g. Feldman and Koffman [1]). These programs compare unfavorably to the graphics of games most students are accustomed to using, and motivation to program in an introductory course may be lost.

Ideally, we would like to be able to have students write programs that have more appealing interfaces, yet do not require a large amount of additional conceptual complexity. In fact, the best case would be to have the student write a program as if it were a simple text-based program, and have the compiler automatically add a graphical interface. Languages that provide overloading, such as Ada 95, allow us to accomplish precisely that.

This paper describes a library, `Graphics_110` (named in honor of our introductory course, CS 110), which, using overloading, replaces the standard I/O libraries in Ada 95: `Ada.Text_IO`, `Ada.Float_Text_IO`, and `Ada.Integer_Text_IO`. By following a simple contract and replacing calls to the standard libraries with calls to `Graphics_110`, the student obtains a program with a Windows-style interface without ever having to worry about the implementation details. Although we use Ada 95 for this paper, the ideas extend to any programming language that provides subtypes and overloading.

The next section describes the “contract” the programmer must follow to use the library and the third section describes the implementation of `Graphics_110`. In each section, we describe how `Graphics_110` was used with a battleship game implemented in our introductory course. The final section presents conclusions and ideas for future work.

## 2 The Contract

`Graphics_110` was designed for simple text-based games. These games share the properties that they interact with the user using text prompts, and display some sort of game board. Common examples of these types of games include tic-tac-toe, role-playing maze games, checkers and battleship. The library provides the user with a 6x80 text window, and a 22x33 grid of bitmaps. (There’s nothing magical about these numbers; they simply fit nicely on our students’ monitors). Text prompts appear in the text window, and dialog boxes are used for input. The game board is displayed in the bitmap grid.

Since we are converting text-based games with no graphics capabilities, we assume that text written to the screen is in one of two forms: either a text sequence (a prompt or informational message), or a display of the game board. The entire game board must be drawn at once. While drawing the game board, the programmer must use defined constant values rather than putting the characters directly. For example, `Put(Item => Ship)` is allowed, but `Put(Item => 'A')` is forbidden. `Ship` is then declared to be a constant `Icon_Type` whose value is `'A'` in the text-based program. (`Icon_Type` is declared to be a subtype of `Character`). An example usage of the library is given in Appendix B.

To convert a program from text-based to Windows-based, one first replaces the `WITH` and `USE` clauses for the `Text_IO` libraries with `WITH` and `USE` clauses for `Graphics_110`. (Although experienced Ada programmers tend to frown on the use of the `USE` clause, we allow it in our introductory course for the sake of brevity). Then, it is

only necessary to comment out the declaration of the subtype `Icon`, and all the constant declarations of that type. These constants will be declared and associated with bitmaps in the instructor-provided `Graphics_110` package.

In summary, we require the student to only make slight changes to meet our contract and then to convert to a Windows-based program. First, we require them to draw the entire game board at once. This does not seem like a large restriction, as in a text-based game, the program usually alternates between prompts and drawing the whole board. Second, we require use of a subtype for everything that will be displayed on the game board. The process for using `Graphics_110` necessitates this, as each of those constants will be redefined in the package provided by the instructor. Finally, after the program is working in a text-based mode, simply by changing the `WITH` and `USE` clauses, and commenting out some declarations, the program will provide a graphical interface.

### 3 Implementation Details

The original version of `Graphics_110` was implemented in Ada 95, using the Microsoft Windows API. To enable broader use of the software, it has been reimplemented using the TASH [3] binding to Tcl/Tk [2]. This section describes initialization, how events are handled, and also what is done for each type of call to a library routine.

Ada 95 permits packages the definition of code bodies within packages that are executed before the main procedure begins. `Graphics_110` uses its code body to open the window, and display a blank grid. By doing this, we keep with our philosophy of requiring as few changes as possible from a text-based program; the student can not accidentally forget to call the initialization routine. (If the library were to be ported to a language that did not have this feature, one could simply keep a boolean variable that keeps track of whether or not initialization has been done. This variable would then have to be checked on each library call, and the initialization routine called if necessary.)

A key assumption that we make is that student programs tend to be I/O bound. As a result, it is not necessary to have the student write an event loop or do polling. Instead each library routine processes any pending events when it is called. Again, by doing this we minimize changes to the student's initial program.

There are four procedures provided by the `Graphics_110` interface (these replace similarly specified procedures from the standard Text I/O library provided with Ada 95): `Put`, `New_Line`, `Get`, and `Skip_Line`. `Put` is overloaded, takes a single argument of type `Integer`, `Float`, `Character`, or `String` and displays it to the screen (we display these in the text window). We have also added an overloaded version of `Put` that takes an icon as an argument (to be displayed in the game board area). `New_Line` advances the cursor to the next line. Ada 95 also has a `Put_Line` procedure, which combines `Put` and `New_Line` for a single string. We provide a similar procedure in `Graphics_110` by simply calling our `Put` procedure followed by our `New_Line` procedure. It has an optional argument that is the number of lines to advance. `Get` is also overloaded, and is used to read either an `Integer`, `Float`, `Character`, or `String` from the keyboard. Its counterpart, `Skip_Line`, is used to read and discard all characters typed at the keyboard until the next time the ENTER key is pressed.

To implement these four procedures, we must keep track of the following global state variables: `LastWasIcon` (did the last call to `Put` have an icon as its argument), `LastWasGet` (was `Get` the last `Graphics_110` routine called), `TextScreen` (copy of all text currently displayed in text window), `BitmapArray` (array of all icons currently displayed on game board), `CurrentTextRow` and `CurrentTextCol` (where cursor is in text window), and `CurrentBitmapRow` and `CurrentBitmapCol` (where last icon was displayed on game board).

For icons, a `Put` is performed by updating the appropriate location in the `BitmapArray`, and then modifying `CurrentBitmapCol`. Since as previously mentioned, our contract requires the student to display the game board all at once, we can use `LastWasIcon` to determine whether or not to return to the upper left corner of the game board. If `LastWasIcon` is false, there has been text displayed, and we must be beginning again in the upper left corner. If incrementing `CurrentBitmapCol` would cause it to exceed its maximum possible value, `New_Line` is called to go to the next row.

Other `Puts` are performed by first (if necessary) creating the string representation of the output (for numbers this is done by calling the appropriate Ada Text I/O library routine) and then calling our `Put` routine for strings. To put a string, we simply modify the array `TextScreen`, placing the characters of the string at the location specified by `CurrentTextRow` and `CurrentTextCol`. The current row is then updated accordingly, and both `LastWasIcon` and `LastWasGet` are set to false. As with icons, `New_Line` is called if necessary to get text wrap.

For a **New\_Line**, if **LastWasIcon** is false, **CurrentTextRow** is incremented, and **CurrentTextCol** is set to 1. Otherwise, **CurrentBitmapRow** and **CurrentBitmapCol** are updated. Following this, **LastWasGet** is set to false, and an internal routine, **UpdateWindow**, is called to process any pending events and redraws the window. **LastWasIcon** remains unchanged.

For a **Get**, a dialog box is displayed. The user then may type the input, and must press ENTER or use the mouse to click “ok.” Graphics\_110 then converts the input to the appropriate type (again using standard Ada Text I/O library routines). **LastWasGet** is set to true and **LastWasIcon** is set to false. Finally the value is returned.

Usually in our students’ programs, following each **Get** is a **Skip\_Line**, which requires the student to press ENTER. Since the dialog box already required the student to press ENTER, this would be redundant. Consequently, if **LastWasGet** is true, the **Skip\_Line** is ignored, and **LastWasGet** is reset to false. There are, however, occasions when **Skip\_Line** is used to pause the program (e.g. if a long text message is being displayed, and the user needs to have control over the speed of scrolling). To accommodate this, if **LastWasGet** is false, **Skip\_Line** displays a dialog box which waits for the user to press ENTER or click “ok” before the program continues.

By keeping the state variables and using them, as above, we are able to obtain the desired Windows behavior using the same sequence of calls as in the text-based program.

#### 4 Conclusions and Future Work

Our implementation of Graphics\_110 allowed us to successfully convert the text-based game written by introductory students to a graphical version. The students have often expressed an interest in being able to write programs containing graphics; however, we have found the amount of detailed coding required to be too great for an introductory course. The Graphics\_110 package enables our students to add a limited amount of graphics and Windows behavior to their programs.

We now have three complete student projects implemented using the Graphics\_110 library, Battleship, Connect Four, and Othello. Each semester, we have the students, working in groups of 2 or 3, implement a game. The students are given the graphics library, and implement the game logic and a computer strategy. The computer strategy procedures are pitted against each other in a tournament. A picture showing a student’s Battleship program during execution is given in Appendix A. Complete sources for the library and these three projects are available at <ftp://ftp.usafa.af.mil/pub/dfcs/carlisle/usafa/graph110/index.html>.

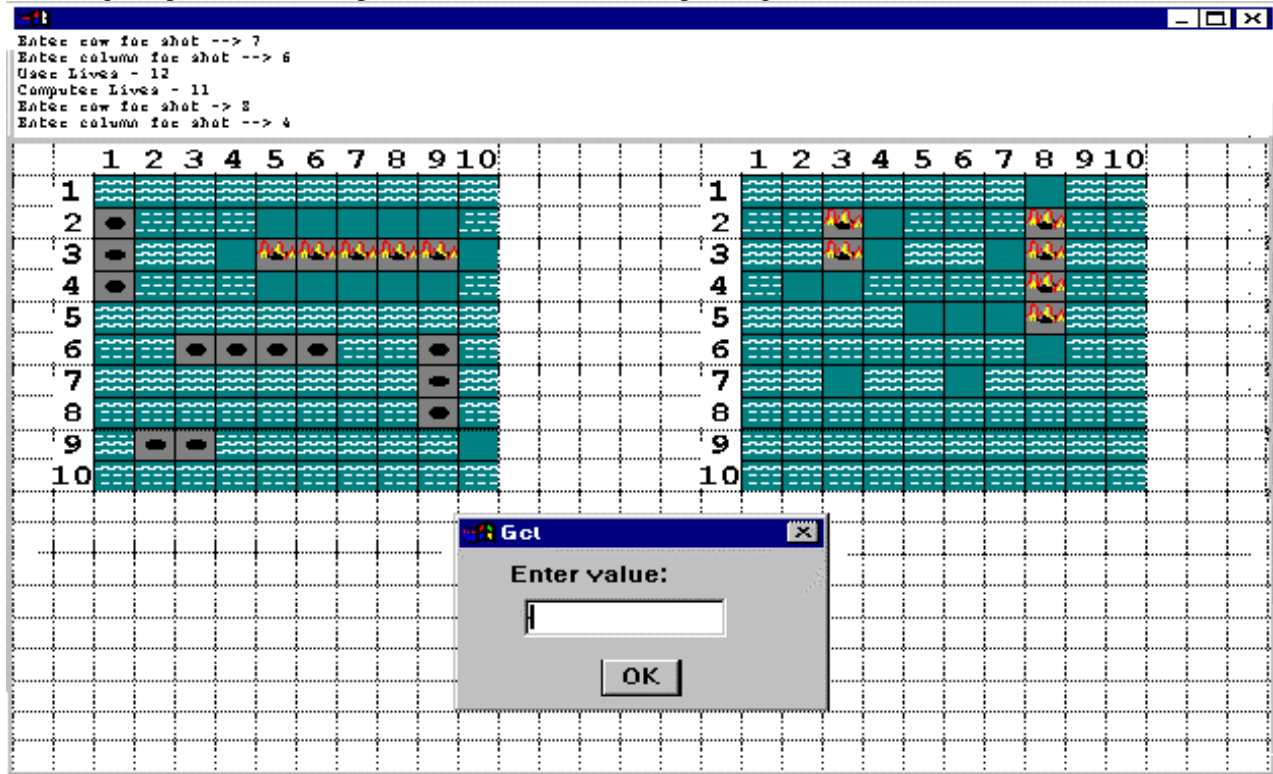
Graphics\_110, while providing a very easy way for students to add graphics to programs, is limited to text games that have some sort of grid. In the future, we would like to explore how to provide a simple abstraction for more complicated graphical programs, such as animations.

#### References

1. Feldman, M. and Koffman, E. *Ada 95: Problem Solving and Program Design, Second Edition*. Addison-Wesley, Reading, Mass., 1996.
2. Ousterhout, J. *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Mass., 1994.
3. Westley, T. TASH: A Free Platform-Independent Graphical User Interface Development Toolkit for Ada. In *Proceedings of Tri-Ada '96*. (1996), 165-178.

## Appendix A

Following is a picture of the Graphics\_110 version of battleship during execution:



## Appendix B

A portion of a Battleship text-based program that is ready to be used with the Graphics\_110 library is shown below. This procedure draws the game board, displaying the computers' ships if the boolean **Display\_Computers** is set to true.

```
SUBTYPE Icon IS Character;

Sea    : CONSTANT Icon := '~';
Hit    : CONSTANT Icon := 'H';
Miss   : CONSTANT Icon := ' ';
Ship   : CONSTANT Icon := 'S';

Blank : CONSTANT Character := ' ';

TYPE Digits IS ARRAY (1..10) OF Icon;
Number : CONSTANT Digits :=
    ('1','2','3','4','5','6','7','8','9','0');

TYPE Board_Array IS ARRAY (1..22,1..33) OF Icon;

PROCEDURE Draw_Boards (
    User_Board      : IN Board_Array;
    Computer_Board  : IN Board_Array;
    Display_Computers: IN Boolean) IS

BEGIN -- Draw_Boards
```

```

-- Leave room for row headings
Put (Item => Blank );
Put (Item => Blank );

-- Display user board col headings
FOR Column IN 1..Num_Columns LOOP
    Put (Item => Number(Column));
END LOOP;

-- Leave space between boards
FOR Column IN 1..6 LOOP
    Put (Item => Blank );
END LOOP;

-- Display computer col headings
FOR Column IN 1..Num_Columns LOOP
    Put (Item => Number(Column));
END LOOP;
New_Line;

FOR Row IN 1..Num_Rows LOOP
    Put (Item => Blank );
    Put (Item => Number (Row));

    FOR Col IN 1..Num_Columns LOOP
        Put(Item=> User_Board(Row,Col));
    END LOOP;

    FOR Col IN 1..5 LOOP
        Put (Item => Blank );
    END LOOP;
    Put (Item => Number (Row));

    FOR Column IN 1..Num_Rows LOOP
        IF Computer_Board(Row,Col)=Ship AND Display_Computers=False THEN
            Put (Item => Sea );
        ELSE
            Put (Item => Computer_Board(Row,Col));
        END IF;
    END LOOP;

    New_Line;

END LOOP;

END Draw_Boards;

```